10:26:39 OCA PAD AMENDMENT - PROJECT HEADER INFORMATION 03/24/94

Active

Project #: C-50-661 Cost share #: C-50-311 Rev #: 4
Center # : 10/24-6-R-7278-0A0 Center shr #: 10/22-1-F7278-0A0 OCA file #:
Work type : RES
Contract#: CCR-9109399 Mod #: ADM REVISION Document : GRANT
Prime #: Contract entity: GTRC

Subprojects ? : N CFDA: 47.070
Main project #: PE #: N/A

Project unit: COMPUTING Unit code: 02.010.300
Project director(s):
STASKO J T COMPUTING (404)853-9386

Sponsor/division names: NATL SCIENCE FOUNDATION / GENERAL
Sponsor/division codes: 107 / 000

Award period: 910815 to 940731 (performance) 941031 (reports)

Sponsor amount	New this change	Total to date
Contract value	0.00	60,000.00
Funded	0.00	60,000.00
Cost sharing amount		88,811.00

Does subcontracting plan apply ?: N

Title: ADAPTING ALGORITHM ANIMATION TECHNIQUES FOR PROGRAM DEBUGGING & TESTING

PROJECT ADMINISTRATION DATA

OCA contact: Jacquelyn L. Tyndall 894-4820

Sponsor technical contact

K.C. TAI
(202)357-7375

NATIONAL SCIENCE FOUNDATION
1800 G STREET, N.W.
WASHINGTON, D.C. 20550

Sponsor issuing office

STEPHEN G. BURNISKY
(202)357-9653

NATIONAL SCIENCE FOUNDATION
1800 G STREET, N.W.
WASHINGTON, D.C. 20550

Security class (U,C,S,TS) : U ONR resident rep. is ACO (Y/N): N
Defense priority rating : supplemental sheet
Equipment title vests with: Sponsor GIT X

Administrative comments -
ISSUED TO EXTEND PERIOD OF PERFORMANCE THROUGH JULY 31, 1994. FINAL REPORT DUE OCTOBER 31, 1994.

GEORGIA INSTITUTE OF TECHNOLOGY
OFFICE OF CONTRACT ADMINISTRATION

NOTICE OF PROJECT CLOSEOUT

Closeout Notice Date 06/14/94

Project No. C-50-661 Center No. 10/24-6-R-7278-0A0

Project Director STASKO J T School/Lab COMPUTING

Sponsor NATL SCIENCE FOUNDATION/GENERAL

Contract/Grant No. CCR-9109399 Contract Entity GTRC

Prime Contract No.

Title ADAPTING ALGORITHM ANIMATION TECHNIQUES FOR PROGRAM DEBUGGING & TESTING

Effective Completion Date 940731 (Performance) 941031 (Reports)

Closeout Actions Required:	Y/N	Date Submitted
Final Invoice or Copy of Final Invoice	N	
Final Report of Inventions and/or Subcontracts	N	
Government Property Inventory & Related Certificate	N	
Classified Material Certificate	N	
Release and Assignment	N	
Other	N	

CommentsLETTER OF CREDIT APPLIES. 98A SATISFIES PATENT REQUIREMENT.

Subproject Under Main Project No.

Continues Project No.

Distribution Required:

Project Director	Y
Administrative Network Representative	Y
GTRI Accounting/Grants and Contracts	Y
Procurement/Supply Services	Y
Research Property Managment	Y
Research Security Services	N
Reports Coordinator (OCA)	Y
GTRC	Y
Project File	Y
Other	N
	N

C 50-661

To NSF Program: _____

APPENDIX VIII

Annual NSF Grant Progress Report

PI Name: John T. Stasko **NSF Award Number:** CCR-9109399

PI Institution: Georgia Institute of Technology

PI Address: College of Computing, Ga. Tech.
Atlanta, GA 30332-0280 **Date:** 12/16/92

I certify that to the best of my knowledge (1) the statements herein (excluding scientific hypotheses and scientific opinions) are true and complete, and (2) the text and graphics in this report as well as any accompanying publications or other documents, unless otherwise indicated, are the original work of the signatories or individuals working under their supervision. I understand that the willful provision of false information or concealing a material fact in this report or any other communication submitted to NSF is a criminal offense (U.S.Code, Title 18, Section 1001.)

Signature: _____

Please include the following information:

1. A brief summary of overall progress, including results obtained to date, their relationship to the general goals of the award and their significance to science;
2. an indication of any current problems or favorable or unusual developments;
3. a brief summary of work to be performed during the next year of support if changed from the original proposal; and
4. any other information pertinent to the type of project supported by NSF or as specified by the terms and conditions of the grant, including a statement describing the contribution of the research in the area of education and human resources development.

If applicable, please attach a copy of any updated human subject or animal subject certification.
[Attach additonal sheets as necessary.]

Understanding and Characterizing Software Visualization Systems*

John T. Stasko, Charles Patterson
Graphics, Visualization and Usability Center, College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280

Abstract

The general term software visualization *refers to graphical views or illustrations of the entities and characteristics of computer programs and algorithms. This term along with many others including* data structure display, program animation, algorithm animation, *etc., have been used inconsistently in the literature, which has led to confusion in describing systems providing these capabilities. In this paper we present a scaled characterization of software visualization terms along aspect, abstractness, animation, and automation dimensions. Rather than placing existing systems into hard-and-fast categories, we focus on unique and differentiating aspects across all systems.*

1 Introduction

A visualization tool provides graphical views of the entities and characteristics of a computer system or program. The purpose of such a visualization tool is stated nicely by Myers, et. al.: "Human information processing is clearly optimized for pictorial information, and pictures make the data easier to understand for the programmer[MCS88]." The two-dimensional format of a picture can provide greater amounts of relevant information more fluently than a stream of text. Programming textbooks reflect this fact when they use the familiar boxes for variables, columns of boxes for arrays, and arrows for pointers. Programmers often draw pictures during the program development process to help transform concepts into actual code. It seems clear, therefore, that the ability to utilize and view graphical program representations can provide aid to software development and understanding. Visualization techniques have already made a significant impact on programming language environments[AB89]. Appropriate animated images also have been used for teaching the purpose and functionality of algorithms[Bro88a, Sta92].

In this paper, we focus on graphical views of computer programs, such as illustrations of variables, code sections, the run-time stack, and program semantics. One term that has become accepted for describing this general area is *program visualization*. Baecker defines the term as, "the use of the techniques of interactive graphics and the crafts of graphic design, typography, animation, and cinematography to enhance the presentation and understanding of computer programs[Bae86]." We shall use the term *software visualization*[PSB92] in this paper as a general encompassing notion for our discussion—It is broader in scope and carries no specific meaning or bias.

These two terms, in addition to many others such as data structure display, program animation, process display, and algorithm animation, have been used to label systems of varying utility. Unfortunately, with so many different names given to so many hybrids of software visualization systems, it is difficult to consistently recognize the purpose of each. Our goal is to characterize the many types of software visualization, both to provide a clearer meaning of the terms in use, and to attempt a structuring of software visualization tasks to guide further work. Lack of clarity in terms is disadvantageous in that when a new system is developed and described by its creators, the capabilities offered by the system are not clear. A precise descriptive scheme provides a framework for designers to describe their work and disseminate information.

Moreover, we seek to provide a better understanding of software visualization systems, why they are important to study and what benefits they can offer to programmers, students, and researchers. We also seek to provide a compendium of systems and capabilities for new researchers entering this area. Many new systems have been introduced recently, and we hope that this survey will coalesce a large body of research into an accessible, succinct form.

Visual programming[Cha87, Shu88] is often confused with software or program visualization, but vi-

*This work supported in part by the National Science Foundation under contract CCR-9109399.

From: IEEE International Workshop on Visual Languages '92, Seattle, WA Sept. 1992.

sual programming differs importantly from the subject matter of this paper. Visual programming involves actual programming through the use of pictures, icons, and graphical entities. The matter addressed herein, however, involves the use of pictures to convey information about programs written in traditional textual languages. A good summary of the distinctions between the two can be found in [Mye90].

In this paper we restrict our focus to visualization. For a good discussion of the new area of software auralization, see [DB92]. We also focus solely on visualization of serial programs and algorithms. Visualizing parallel program involves all the serial visualization issues we discuss in this paper plus new problems unique to concurrent programming.

2 Characterizing Software Visualizations

Taxonomies of software visualization systems already exist. Myers has developed a classification scheme using two axes: whether the systems illustrate the code, data, or algorithm of a program, and whether they are dynamic or static[Mye90]. Singh presents a similar scheme[Sin90]. Price, Small, and Baecker[PSB92] use a comprehensive 30-category taxonomy to describe software visualization systems. On the surface, our characterization has similarities to these—we utilize four classifying dimensions with two corresponding closely to Myers' two dimensions mentioned above. But our primary purpose is not to place existing systems into labelled categories. Rather, we seek to show how different systems exhibit varying levels of the four dimensions we have identified. That is, we use features of existing systems to illustrate and clarify the more general fundamental concepts of software visualization systems. We seek to supplement the taxonomies above with a discussion of the fundamental issues and choices confronting software visualization researchers.

The characterization scheme we propose contains four dimensions:

- Aspect
- Abstractness
- Animation
- Automation

2.1 Aspect

Software visualization systems usually focus on a different *aspect* of a program to be visualized. For example, a system may supply visualizations of program text, data structures, run-time program state, control flow, or algorithmic methods. This dimension most closely represents the purpose of the visualization—why the visualization is being created and what parts of the program are being emphasized. The aspect dimension also contrasts from the others in our scheme in its discrete, rather than continuous, nature.

The simplest aspect level of software visualization is just an enhanced presentation of program text. Two widely used early visualization techniques for program text are flowcharts and Nassi-Shneiderman Diagrams[NS73]. The Greenprint system[BEP80] provides a nested block and box graphical representation of a program that is placed beside program text to illustrate control flow. The SEE system[BM90] uses human factors knowledge and typography techniques to display C programs. Debuggers often show the text of programs' procedures as they execute, with line by line highlighting.

Moving beyond purely textual views, some systems provide views of the data and data structures in programs. One of the first general purpose *data structure display* systems, Incense[Mye83], generates a view of user-specified data structures during debugging. A follow-up system, MacGnome[MCS88], focuses on providing simple canonical Pascal data structure views for novice programmers.

Some systems provide views of program aspects beyond pure data structures. For instance, a system may include views of flow-of-control such as a rendering of program subroutines as icons, a call-graph view, run-time stack view, etc., in addition to data structure views. We call these types of systems *program state visualization* systems. The Pecan system[Rei85] contains a large set of views including symbol table, data type, stack, flowgraph, and expression displays. The PV system[B+85] provides program structure, control flow, and data views, but it also includes views of important phases of the software engineering lifecycle such as diagrams of system requirements.

The aspect of a program being visualized depends upon the underlying programming paradigm of its language. The aspects being visualized in the above systems naturally correspond to imperative languages. For functional languages, other aspects of a program are important. The KAESTLE and Fooscape systems[BFN86] present list structures and the network of function calls within a LISP program. Lieberman[Lie89] uses a unique three-dimensional representation to visualize LISP program structures. In a logic programming environment, the pertinent clauses and goals are critical aspects to be visualized. The

Transparent Prolog Machine[EB88] is an interpreter that visually depicts Prolog program traces.

Yet a further level of program display provides views of the underlying algorithm or higher-level strategy of a program. *Algorithm visualization* systems are systems that provide visual depictions of the purposeful operations, methodologies, and tactics that programmers utilize in their programs. These types of systems are not concerned with the details of a particular implementation in a programming language. Rather, they focus on the fundamental methods utilized to solve a problem. Their displays are inherently semantic, thus differing from views of isolated data.

A graphical view of a comparison sort provides a good example of an algorithm visualization. The view may represent array elements as rectangles and highlight the rectangles when array values are compared prior to a possible exchange. This graphical action involves a mapping from the meaning of the program to the display. Program state visualization systems display code and its syntactic structure, such as scope, but they stop short of showing views of the actual task being performed by the code. One of the first examples of algorithm visualization is the film *Sorting Out Sorting*[BS81], generally accepted as a motivating factor for this research area. The Balsa system[Bro88a] is the prototype for algorithm visualization systems with its high-quality imagery, multiple views, and scripting facilities. Balsa inspired subsequent systems such the Smalltalk based system Animus[Dui86], and ANIM[BK91], a system for building simple algorithm visualizations in a UNIX environment.

Algorithm visualization systems often include data structure and program state visualization capabilities. The LOGOmotion system[BB90], for example, provides default variable-update and procedure events in addition to supplying a programmable visualization language. No one particular existing system, however, completely spans all ranges of our aspect dimension.

2.2 Abstractness

Even though software visualization systems may display views of the same aspect of a program, the level of *abstractness* at which the view is presented may vary widely. For example, a data structure display system may render three integer variables named `hours`, `minutes`, and `seconds` as three rectangular boxes containing the individual values as text strings. However, another view of this data structure could display the data in the form of a clock face with appropriate hour, minute, and second hands.

One characterization of the abstractness of a program view is whether the display is isomorphic to the program components it represents[Bro88b]. That is, could a data structure be rebuilt from its graphical representation as easily as the representation is created from the data structure?

Algorithm visualizations, as discussed in the previous subsection, typically go beyond isomorphic mappings of program data or code to graphical representations of program semantics. Consequently, algorithm visualizations inherently provide a high level of abstractness, and they have even been defined accordingly[Sta90]. For instance, a visualization of a program performing an exhaustive search might contain a bar representing the number of unsuccessful search attempts it has made. As more unsuccessful attempts accumulate, the bar grows larger. This idea of "incorrect attempts" may not be represented anywhere in the program, but it has semantic meaning with respect to the program's purpose.

To help understand the use of abstractness by software visualization systems, we introduce the concept of *intention content*, the semantics or meaning behind otherwise context-free data and code. Given a task or entity to be visualized, the intention content of the visualization is the level of knowledge about the task's purpose required to map the task to the visualization. Greater amounts of intention content support displays that are more informative and that are more abstract. Also, a greater level of intention content requires a programmer (of the software being viewed) to provide the visualization system with more information on the details of what to display.

For example, consider a view of a sort in an algorithm visualization system. Without any intention content, the display could present an array of values as an indexed list. After each execution cycle or at programmer specified times, the values on the screen in the array would update to match the current state of the process. Actually, this will work with any array, whether or not sorting is involved. More useful didactically are the values of the array displayed as bars of varying height, taller bars for larger values, which trade places as the sort algorithm swaps them. However, this display cannot be standard for all arrays. For instance, an array of indices into another array is meaningless as a row of bars. If a high level of intention content is to be used, the programmer's purpose is necessary and must be supplied to the system.

The algorithm animation system ZEUS[Bro91] provides program event views that vary in their level of abstractness. It is possible to look at program events as a transcript (just a textual list), as a control panel (buttons for the vents with graphical widgets for the

parameters), and as the abstract graphical imagery and actions representing the event's execution.

Intention content is important in data structure display as well. The lowest intention content level for data structures is represented by the classic box-and-arrow diagram, commonly shown in data structures and programming textbooks. Integers, reals, booleans, etc., each have a distinct box representation complete with name and value. Composite data structures, such as records and structures, are built by using an encompassing box representation around their elements' boxes. Pointers or addresses are represented by arrows to the objects they reference. (Conceptually, the very lowest intention content level would be a string of binary bits of length equal to the computer space used by a data structure, but this is almost always below our needs.)

Figure 1 shows a record data structure, consisting of an integer and an array, interpreted under differing intentions. Each view is a possible interpretation of what the data structure signifies. The view in Figure 1a shows the classic interpretation mentioned above, at the lowest intention content level.

Figure 1b shows a data structure view with more intention content than the one in Figure 1a. Here, the array is shown as a bar graph with the bars scaled according to the accompanying array element's value. An even more specific view, Figure 1c shows a pie graph of the array made possible since the data structure visualization system contains the knowledge that the values represent percentages of components in a whole, such as the percent of elements in a chemical compound. Finally, Figure 1d shows the array data structure and integer value interpreted as a stack. The single integer, which to this point has been an unrelated part of the record containing the array, is the top pointer for the stack. This type of view exhibits high levels of intention content and abstractness.

Some systems such as the data structure display system Incense[Mye83], support the creation of both low-level concrete views and highly abstract views such as the clock face described above. The system can generate the low-level, canonical views automatically, but it requires programmer assistance (writing display procedures) to generate abstract views.

2.3 Animation

Our third classification dimension describes the dynamics or *animation* shown in software visualization systems. Unfortunately, the term "animation" also has been loosely applied in the past, and a large variety of systems claim to provide animation capabilities.

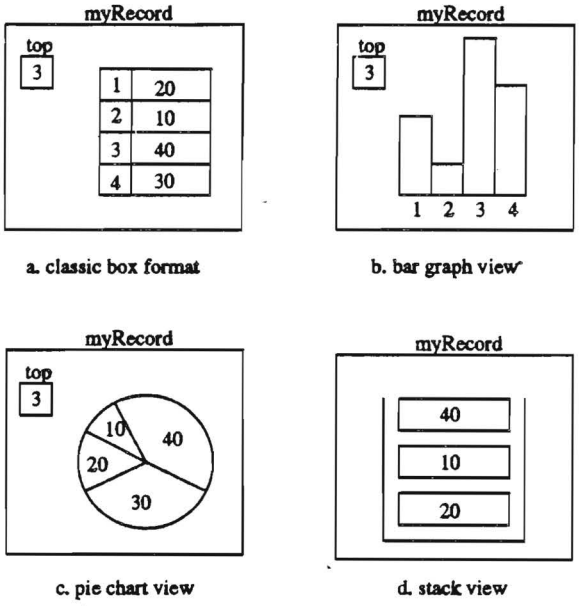

Figure 1: Illustrating various intention content levels by altering the representation of a data structure.

For example, actions such as simply highlighting lines of code as they are executed, altering the boundary style of a graphical object, or changing color intermittently have been called animation.

Fundamentally, animation consists of the rapid sequential display of pictures or images, with the pictures changing gradually over time. These pictures are the frames of the animation. If the imagery's changes from frame to frame are small enough and the speed of displaying the frames is fast enough, the illusion of continuous motion is achieved.

We consider *data structure animation* systems and *program state animation* systems to be systems which support repeated display of data structure and program state visualizations, respectively, with changes in view sufficient in both content and time to provide a viewer with the essence of how the data and program transform continuously throughout execution.

As an example of what we mean, consider views of a linked list data structure being built. If, when a new node is added, it is shown immediately at its correct position in the list (say a horizontal row of nodes) this would be considered data structure visualization. But if a view displays a new node being allocated in a special heap memory area, then the node slides over to its correct position in the list, this would be considered a data structure animation.

An even more rigid specification is proposed for creating animations of algorithms. To motivate the criteria, we introduce the notion of a *valid configuration*. A valid configuration of a program is a state (data values, context, point of control) of the program that involves semantic meaning and that is reachable during execution. As a program executes, it transforms from valid configuration to valid configuration. These configurations can be at a fine-grain level such as after each line of execution, or at a higher level such as in a sorting program, after each exchange operation. Valid configurations identify program contexts that make sense in terms of the program's purpose and functionality.

Because an algorithm connotes meaning beyond simple data objects, the transitions between valid configurations, in addition to the configurations themselves, take on added importance. One of the primary goals of animating programs is to illustrate not just the set of states a program reaches during execution, but how the transformations between states occur. In order to incorporate this fact, we consider an *algorithm animation* system to be a system that illustrates a program's behavior by both repeatedly displaying graphical images corresponding to valid configurations and displaying sequences of graphical images that correspond to states "in-between" those configurations. Scene display should occur at a sufficiently brisk pace to provide the illusion of continuous motion.

Essentially, the views that are shown between valid configurations denote configurations that are never realized and carry no semantic meaning. They are produced strictly for aesthetic reasons and for illustrating how transitions between valid configurations occur. Perhaps an example best illustrates this concept.

Consider the graphical depiction of a sorting program, discussed earlier, that represents the program's data elements as a row of rectangular images. The two scenes in Figure 2 illustrate two consecutive valid configurations that we might reach during execution of the program. In the second scene, elements 2 and 3 have exchanged their positions from the first scene. The repeated display of such configurations reached during execution does not constitute an algorithm animation according to our characterization. Rather, it would be an algorithm visualization because no intermediate scenes between valid configurations were presented. (Note, this is an algorithm visualization with extremely low intention content, which in fact, could be considered a sophisticated data structure visualization. We use it here to illustrate a point.) On the other hand, Figure 3 shows a superimposed sequence of frames from what we would consider an

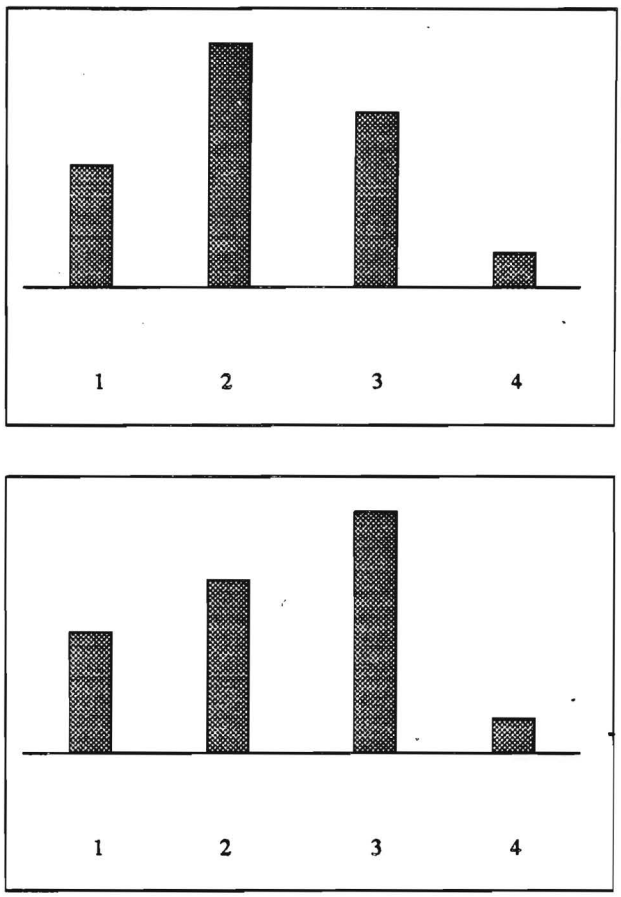

Figure 2: Visualizations of consecutive valid configurations from a bubblesort algorithm.

algorithm animation. In it, the rectangles assume a set of slightly altered positions between the two in Figure 2. If the intermediate scenes were displayed quickly enough, they would present a definite illusion of motion. The important concept here is that all of the intermediate scenes represent program states that never really exist and have no meaning in terms of the program context. They are purely artificial states, created for the viewing aesthetics of the animation.

An animation of the Towers of Hanoi problem (often used to teach recursion) with disks moving between the separate pegs is another example of a visualization in which showing a series of intermediate frames of the disks' movements is absolutely critical for understanding. Simply presenting a rapid sequence of frames with the disks at their new end-positions (without intermediate movement presented) would be extremely difficult to follow and comprehend.

Brown has characterized specific imagery in algorithm animations along three dimensions: transforma-

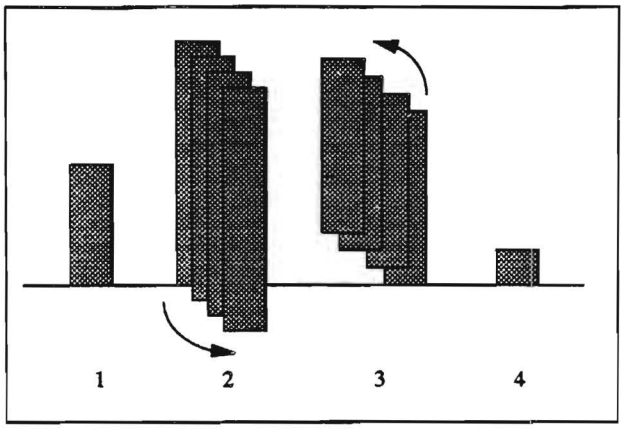

Figure 3: Bubblesort algorithm animation with "in-between" configurations shown.

tion, persistence, and content[Bro88b]. The animation dimension of our scheme coincides similarly with his transformation dimension. Our notion of characterizing animation by the artificial program states presented between valid program configurations, however, helps to clarify the distinction between software visualizations and animations.

One of the earliest systems to recognize the importance of smooth transitions generates algorithm animations of Smalltalk programs by monitoring message passing[LD85]. Later software visualization systems such as Tango[Sta90] provide explicit mechanisms to help produce the in-between configuration views that typify animations. In Tango, view designers develop animations using high-level primitives that hide low-level graphics details. It is still possible, however, to generate more traditional visualization views without the in-between frames in these systems. In fact, some views are more informative when explicit animation (as we characterize it) is not utilized. For example, views involving large data sets and data structures often do not require explicit animation. In these views, the extra frames from animation may slow down the presentation and hinder understanding.

2.4 Automation

We consider the level of *automation* provided for developing software visualizations to be another characterizing factor in systems. Automation levels can range from totally automatic views generated as a program executes to views requiring explicit programmer design and implementation effort, as well as specification of the appropriate trigger points in the program.

Data structure display system views are usually generated automatically, without explicit programmer support. That is, an execution monitor or debugger examines a program at a specific moment and provides information to generate graphical views of the data structures. This capability, with no turnaround time for view design, is necessary for time-intensive tasks such as debugging. Often, systems support automatic generation of displays, but they also permit viewers to modify the display as desired. The GDBX system[Bas85] provides canonical box-and-arrow displays of Pascal and C programs. It allows viewers to reposition or eliminate data structure views as desired during a debugging session. GELO[RMD89] also includes predefined data views, but it allows users to graphically specify specialized data type displays using topological constraints.

It is possible to think of data structure displays that would be very difficult to generate automatically too. The clock face view discussed earlier would be impossible to automatically generate without some designer assistance and direction. This fact illustrates that our abstraction and automation dimensions usually exist in an inverse relationship. Creating software visualization views with high levels of abstractness involves a great deal of intention content and simply requires a priori design support.

Program state views such as those of the call graph, run-time stack, or text code can be generated automatically by a software visualization system. Again, these views are extremely useful for debugging which requires little of no view set-up time. For instance, VIPS[ISO87] generates multiple run-time views of Ada programs, including data, block structure, and debugger interaction windows. Often, the most difficult part of building a system to display program state is not the generation of the graphics, but the acquisition of the run-time execution data and information driving the graphics. This may require low-level coding that examines compiler information, symbol table access, or debugger internals.

As discussed earlier, algorithm visualization systems provide visual depictions of the semantic notions and abstractions used in computer programs and processes. Algorithm visualizations can display information that is not immediately evident or that cannot be automatically deduced by examining the program state during execution. That is, algorithm visualizations require high levels of intention content from a programmer. They are usually hand-crafted, user-conceptualized views of what is "important" about a program, so they require a designer to specify and implement the graphics that accompany a program.

Consequently, algorithm visualizations (and particularly animations), virtually by definition, exhibit a

of automation. Recently, systems providing n views without explicit end-designer support peared[HWF90], but they are restricted to spegorithm domains and they require considerable ler crafting. Brown makes strong arguments under current constraints, explicit programmer gn for algorithm animations is typically required l desirable[Bro88b].

Nevertheless, recent algorithm animation work has ocused on reducing the burden of view design and implementation. These efforts still provide a designer with artistic freedom, but they strive to provide tools which make view development easier and more fun. The Aladdin system[HHR89] uses a declarative mechanism to specify view layout. Programmers interleave graphical specifications throughout a program, which is then executed to generate the visualization. The Gestural system[Dui87] and the Dance animation editor[Sta91] both allow designers to "visually program" their desired visualization via direct manipulation. ZEUS[Bro91] includes a graphical editor for designers to specify how view objects should look.

3 Summary

We have presented, under the area of software visualization, a characterization along the four dimensions of aspect, abstractness, animation, and automation. We clarified the meaning of the dimensions by illustrating how specific system aspects fit within them. The notion of intention content was introduced to help explain the abstractness dimension. We also clarified the differences between visualization and animation by making the distinction that animation presents views of a program between its valid configurations.

References

[AB89] Allen L. Ambler and Margaret M. Burnett. Influence of visual technology on the evolution of language environments. *Computer*, 22(10):9–22, October 1989.

[B+85] Gretchen P. Brown et al. Program visualization: Graphical support for software development. *Computer*, 18(8):27–35, August 1985.

[Bae86] Ronald M. Baecker. An application overview of program visualization. *Computer Graphics: SIGGRAPH '86*, 20(4):325, July 1986.

[Bas85] David B. Baskerville. Graphic presentation of data structures in the DBX debugger. Technical Report UCB/CSD 86/260, University of California at Berkeley, Berkeley, CA, October 1985.

[BB90] Ronald M. Baecker and J. W. Buchanan. A programmer's interface: A visually enhanced and animated programming environment. In *Proceedings of the 23rd Hawaii International Conference on System Sciences*, pages 531–540, Kailua-Kona, HI, January 1990.

[BEP80] L. A. Belady, C. J. Evangelista, and L. R. Power. GREENPRINT: A graphics representation of structured programs. *IBM Systems Journal*, 19(4):79–90, 1980.

[BFN86] Heinz-Dieter Bocker, Gerhard Fischer, and Helga Nieper. The enhancement of understanding through visual representations. In *Proceedings of the ACM SIGCHI '86 Conference on Human Factors in Computing Systems*, pages 44–50, Boston, MA, April 1986.

[BK91] Jon L. Bentley and Brian W. Kernighan. A system for algorithm animation. *Computing Systems*, 4(1), winter 1991.

[BM90] Ronald M. Baecker and Aaron Marcus. *Human Factors and Typography for More Readable Programs*. Addison-Wesley, Reading, MA, 1990.

[Bro88a] Marc H. Brown. Exploring algorithms using Balsa-II. *Computer*, 21(5):14–36, May 1988.

[Bro88b] Marc H. Brown. Perspectives on algorithm animation. In *Proceedings of the ACM SIGCHI '88 Conference on Human Factors in Computing Systems*, pages 33–38, Washington D.C., May 1988.

[Bro91] Marc H. Brown. ZEUS: A system for algorithm animation and multi-view editing. In *Proceedings of the IEEE 1991 Workshop on Visual Languages*, pages 4–9, Kobe Japan, October 1991.

[BS81] Ronald M. Baecker and David Sherman. Sorting out sorting. 16mm color sound film, 1981. Shown at SIGGRAPH '81, Dallas TX.

[Cha87] Shi-Kuo Chang. Visual languages: A tutorial and survey. *IEEE Software*, 4(1):29–39, January 1987.

[DB92] Christopher J. DiGiano and Ronald M. Baecker. Program auralization: Sound enhancements to the programming environment. In *Graphics Interface '92*, pages 44–52, Vancouver, B.C., May 1992.

[Dui86] Robert A. Duisberg. Animated graphical interfaces using temporal constraints. In *Proceedings of the ACM SIGCHI '86 Conf. on Human Factors in Computing Systems*, pages 131–136, Boston, MA, April 1986.

[Dui87] Robert A. Duisberg. Visual programming of program visualizations. A gestural interface for animating algorithms. In *IEEE Computer Society Workshop on Visual Languages*, pages 55–66, Linkoping, Sweden, August 1987.

[EB88] M. Eisenstadt and M. Brayshaw. The transparent prolog machine (TPM): An execution model and graphical debugger for logic programming. *Journal of Logic Programming*, 5(4):1–66, 1988.

[HHR89] Esa Helttula, Aulikki Hyrskykari, and Kari-Jouko Räihä. Graphical specification of algorithm animations with Aladdin. In *Proceedings of the 22nd Hawaii International Conference on System Sciences*, pages 892–901, Kailua-Kona, HI, January 1989.

[HWF90] Robert R. Henry, Kenneth M. Whaley, and Bruce Forstall. The University of Washington illustrating compiler. *Sigplan Notices: SIGPLAN '90*, 25(6):223–233, June 1990.

[ISO87] Sadahiro Isoda, Takao Shimomura, and Yuji Ono. VIPS: A visual debugger. *IEEE Software*, 4(3):8–19, May 1987.

[LD85] Ralph L. London and Robert A. Duisberg. Animating programs using Smalltalk. *Computer*, 18(8):61–71, August 1985.

[Lie89] Henry Lieberman. A three-dimensional representation for program execution. In *Proceedings of the 1989 IEEE Workshop on Visual Languages*, pages 111–116, Rome, Italy, October 1989.

[MCS88] Brad A. Myers, Ravinder Chandhok, and Atul Sareen. Automatic data visualization for novice Pascal programmers. In *IEEE Computer Society Workshop on Visual Languages*, pages 192–198, Pittsburgh, PA, October 1988.

[Mye83] Brad A. Myers. A system for displaying data structures. *Computer Graphics: SIGGRAPH '83*, 17(3):115–125, July 1983.

[Mye90] Brad A. Myers. Taxonomies of visual programming and program visualization. *Journal of Visual Languages and Computing*, 1(1):97–123, March 1990.

[NS73] I. Nassi and B. Shneiderman. Flowchart techniques for structured programming. *ACM SIGPLAN Notices*, 8(8):12–26, August 1973.

[PSB92] Blaine A. Price, Ian S. Small, and Ronald M. Baecker. A taxonomy of software visualization. In *Proceedings of the 25th Hawaii International Conference on System Sciences*, volume II, pages 597–606, Kauai, HI, January 1992.

[Rei85] Steve P. Reiss. Pecan: Program development systems that support multiple views. *IEEE Transactions on Software Engineering*, SE-11(3):276–285, March 1985.

[RMD89] Steven P. Reiss, Scott Meyers, and Carolyn Duby. Using GELO to visualize software systems. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software*, pages 149–157, Williamsburg, VA, November 1989.

[Shu88] Nancy C. Shu. *Visual Programming.* Van Nostrand Reinhold, New York, NY, 1988.

[Sin90] Gurminder Singh. Graphical support for programming: A survey and taxonomy. In T. S. Chua and T. L. Kunii, editors, *CG International '90*, pages 331–359. Springer-Verlag, 1990.

[Sta90] John T. Stasko. TANGO: A framework and system for algorithm animation. *Computer*, 23(9):27–39, September 1990.

[Sta91] John T. Stasko. Using direct manipulation to build algorithm animations by demonstration. In *Proceedings of the ACM SIGCHI '91 Conference on Human Factors in Computing Systems*, pages 307–314, New Orleans, LA, May 1991.

[Sta92] John T. Stasko. Animating algorithms with XTANGO. *SIGACT News*, 23(2):67–71, spring 1992.

Annual Report
NSF RIA Award CCR-9109399

Adapting Algorithm Animation Techniques for Program Debugging and Testing

John T. Stasko

College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280

The work on my Research Initiation Award proceeded quite well during the first year of the grant. Briefly reviewing, my project is addressing the application of algorithm animation techniques toward visual debugging. Algorithm animation is the use of dynamic graphics to help illustrate or explain computer algorithms and programs. Algorithm animations have been used as pedagogical aids to accompany classroom lectures and to help better understand what the algorithm is doing. The visualizations and animations provide a concrete depiction of the algorithm's methodology and operations.

My research is to investigate how these program illustration techniques can be adapted to help assist software development and debugging. Unfortunately, algorithm animations currently require lengthy development periods including intensive graphics coding. This precludes their general application to debugging which is a time-intensive activity. So I am seeking ways to adapt these techniques for application by users without textual coding.

To begin the research I characterized the different types of software visualization systems that have been developed[2]. Algorithm animation is simply one category of software visualization. Another category, data structure display, exhibits certain characteristics similar to the goals of this project. I studied some of the existing data structure display systems and am incorporating their methods into the system I'm developing. I published a paper at the 1992 Visual Languages Workshop which described the characterization of software visualization systems I carried out. The characterization included four dimensions: aspect, abstractness, animation and automation.

My research then proceeded to identify the most common graphical objects and actions used in current algorithm animation systems. A graduate student, Sougata Mukherjea, and I studied many algorithm animations developed with the XTango algorithm animation system, and we found some interesting results: A small kernel of graphical objects, lines, rectangles, circles, and text, was used. Also, a small kernel of animation actions was used: move to a new position, change color, change visibility, flash, and exchange.

We then began the development of a software system that would smoothly incorporate both debugging capabilities and the algorithm animation techniques just mentioned above. This past summer we were able to develop a working prototype of the system. (We call it

Lens.) Lens allows programmers to view their program's source and drop animation commands down at various positions. When the program's execution reaches those points, the animation command defined there will commence. The tool includes a graphical editor to allow programmers to describe the appearance of their program simply by drawing presentations for data structures. Commands such as move and color are chosen from menus, and the user is prompted to enter their parameters through dialog choices.

We submitted a description of the system prototype to the 1993 International Conference on Software Engineering, and we recently learned that it has been accepted for publication[1].

Our research work is proceeding roughly on schedule and I anticipate no major problems during the second year of the award. I do not anticipate any major changes from the plan either. We will continue to develop and refine the current system prototype, and we hope to experiment with its use by other members of the academic community here.

I have attached copies of the two papers mentioned above that have resulted from research undertaken through this award. Please note that the second paper is only a draft form and not the final camera ready version.

References

[1] John T. Stasko and Sougata Mukherjea. Applying algorithm animation techniques for program tracing, debugging, and understanding. In *Proceedings of the 15th International Conference on Software Engineering*, Baltimore, MD, May 1993. (To appear).

[2] John T. Stasko and Charles Patterson. Understanding and characterizing software visualization systems. In *Proceedings of the IEEE 1992 Workshop on Visual Languages*, pages 3–10, Seattle, WA, September 1992.

DRAFT

Applying Algorithm Animation Techniques for Program Tracing, Debugging, and Understanding

Sougata Mukherjea
John T. Stasko[1]

College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280

E–mail: {sougata, stasko}@cc.gatech.edu

Abstract

Algorithm animation, which presents a dynamic visualization of an algorithm or program, primarily has been used as a teaching aid. The highly abstract, application-specific nature of algorithm animation requires human design of the animation views. We speculate that the application-specific nature of algorithm animation views could be a valuable debugging aid for software developers as well. Unfortunately, if animation development requires time-consuming design with a graphics package, it will not be used for debugging, where timeliness is a necessity. We have developed a system called Lens that allows programmers to rapidly (in minutes) build algorithm animation-style program views without requiring any sophisticated graphics knowledge or coding. Lens is integrated with a system debugger to promote iterative design and exploration.

Keywords: Algorithm animation, debugging, software visualization, program development

[1]Author for contact

1 Introduction

People invariably have a difficult time understanding abstract concepts or processes. One way to improve understanding is to provide specific examples, possibly using pictures to make an abstract concept more concrete. The expression "Seeing is believing" relates that what we can visualize, we can grasp and understand.

This notion can be applied to software understanding and the use of graphics and visualization to depict computer algorithms and programs. Software visualization provides concrete representations to previously inanimate, abstract entities that have always been, and most likely always will be, relatively difficult to understand. The use of graphics for illustrating software was originally called *program visualization*[Bae86, Mye90], but more recently the term *software visualization*[PSB92, SP92] has been favored. The term "software visualization" is better because it is more general, encompassing visualizations of data structures, algorithms, and specific programs. In fact, software visualization research primarily has concentrated on two different subtopics: data structure display and algorithm animation.

Data structure display systems such as Incense[Mye83], GDBX[Bas85], and VIPS[ISO87, SI91] illustrate particular data structures within a program, showing both the values and interconnections of particular data elements. These systems automatically generate a view of a data structure when the user issues a command to do so. Other systems such as Pecan[Rei85] provide additional program state views of control flow, the run-time stack, and so on. Data structure display systems main application has been for debugging and software development. Commercial systems such as Borland C for PCs and CodeVision for SGI workstations even provide rudimentary data structure display capabilities within a software development environment.

Algorithm animation, the second main subarea of software visualization, provides views of the fundamental operations of computer programs, concentrating more on abstractions of behavior than on a particular program's implementation[Bro88]. The views presented are much more application-specific than the generic views of data structure display systems. The movie *Sorting out Sorting*[BS81] motivated many of the subsequent algorithm animation software systems that have been developed, including Balsa[BS85], Animus[Dui86], Movie[BK91], and Tango[Sta90]. In all these systems, a developer first designs a visual presentation for an algorithm, then s/he implements the visualization using a support graphics platform. The primary use of algorithm animations has been for teaching and instruction.

Because algorithm animation views are complex user-conceptualized depictions, they cannot be created automatically from a "black-box" program illustrator. Rather, an animation designer crafts a particular view that is specifically tailored for a program. Consequently, designing algorithm animations is time-intensive and usually restricted to already working programs. That is, a designer utilizes a fully functional working program, designs a set of animation routines for its visualization, and maps the program to the corresponding routines. The resulting views can then be used as instructional aids for illustrating the program's methodologies and behaviors. Unfortunately, the time and effort required to develop animations is considerable enough to limit their use to pedagogy and preclude their use in software development and debugging. A programmer will not use a tool for debugging whose development time outweighs that to simply debug a program with traditional

text-based methods.

This fact is unfortunate, however, because algorithm animations could offer key benefits to program debugging and testing. The use of pictures to illustrate how programs work has always played an important role in software engineering. Programmers, in designing code, often implicitly construct a mental model of their program, how it should function, how it utilizes data, and so on. It is much easier for people to think in terms of these higher level abstractions (the big picture) than to try and comprehend how all of the individual operations and data work in conjunction—particularly so for larger systems. Data structure display systems, which have already been utilized in debuggers, can only offer views of raw data; the notion of an overall picture does not exist there. Algorithm animations, conversely, offer views of a program's application domain and semantics. These types of views can be critical for determining why a program is not performing in its desired manner.

In particular, the use of animation is extremely important because programs are fundamentally dynamic. Illustrating program data and states is useful for understanding, but illustrating how the program changes from state to state and evolves over time is even more helpful. Consider developing a computational geometry program, a quicksort, a particle chamber simulation, or a graph colorability algorithm. Would it not be extremely advantageous to have a dynamic visualization of the program to watch during program testing in order to *see* how it is working and to help identify erroneous program actions?

Particular systems have taken steps toward this merge of data structure display and algorithm animation. The data structure display system Incense[Mye83] allows developers to design their own abstract views of data structures, such as showing a clock face to represent integer values *hours* and *minutes*. This design, however, requires writing the low-level graphics code to implement the view.

The algorithm animation system Movie[BK91] focuses on rapid development of relatively straightforward algorithm visualizations. The system provides a few simple commands such as which can be used to quickly develop a visualization of a program in order to understand it better. Programmers still must learn the system's commands, however, and the system does not support smooth, continuous animation effects.

The Gestural system[Dui87] supports purely graphical animation development on Smalltalk programs, but its only images are black rectangles and its only actions were movements following mouse-traced paths.

The Dance system[Sta91] allows developers to graphically build algorithm animations, then it generates the corresponding Tango[Sta90] animation code to carry out the specified actions. Unfortunately, programmers still must learn the underlying animation paradigm of Tango to develop views of algorithms or programs. Also, animation designs cannot be incrementally tested. The code must be generated, compiled, and run, but the design cannot be read back into the system for modifications.

The University of Washington Program Illustrator[HWF90] truly requires no designer input to generate an algorithm animation. It analyzes program source code and produces its own abstract depiction. The system was only developed to analyze sorting and graph algorithms, however, using one particular style of view for each. Building an automatic "black-box" algorithm animation style viewing tool for arbitrary programs appears to be an impossible task because of the infinite number of different possible algorithms and de-

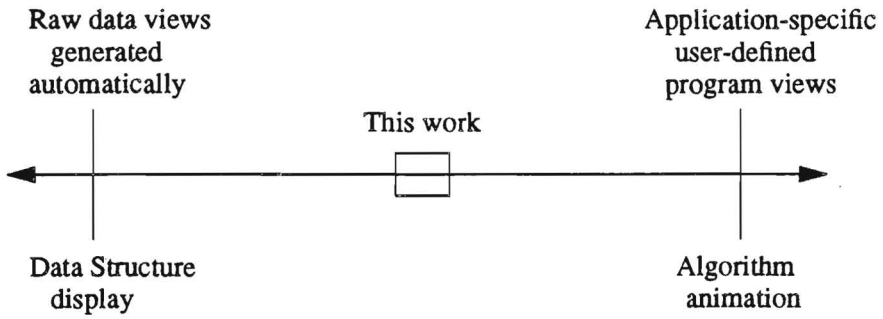

Figure 1: This work bridges the different areas of data structure display and algorithm animation, seeking to gain the benefits of both.

pictions.

Our work seeks to bridge the two domains of data structure display and algorithm animation as illustrated in Figure 1. We want a system that can provide application-specific animation views for debugging purposes. Unlike the UWPI system, we still want programmers to design their own animations. We do not want to require the programmers to need to learn a graphics toolkit and write code using it, however. We also want our tool to work in conjunction with a debugger so that animation can be incrementally developed without going through the edit-compile-run cycle.

This work addresses the tail-end of the software development pipeline, when a designer has already written portions of, or perhaps all of, a target program or system. It is best-suited for high-level debugging, program testing, and refinement, not low-level debugging typically focusing on correcting inadvertent lexical or syntactic misuses. Debugging has been characterized as the acquisition of clues to help programmers generate hypotheses on why a program is not working[Gou75, KA87]. Viewing program execution through dynamic visualizations is a clear, practical way to generate the clues which might be critical for correcting programs.

We have developed a system called Lens that allows programmers to rapidly develop animations of their programs. Lens supports application-specific semantic program views as seen in many algorithm animation systems, but it does not require graphics programming. Lens is integrated with a system debugger to support iterative testing and refinement. In the remainder of this article, we describe the conceptual model on which Lens is based, we illustrate how program animations are built with Lens, and we describe some of the implementation challenges the system presented.

2 Identifying the Essential Components of Program Animations

In building the algorithm animation debugging environment, we sought to provide a palette of commonly-used operations to developers for direct invocation, rather than forcing devel-

opers to write animation description code. We also sought to provide a graphical editor for defining the appearance of program variables. Most importantly, we wanted to keep the lists of operations and graphical editing operations to a minimum of often-used directives. Rather than building a comprehensive environment which could support any animation but which also included many rarely used directives, we sought to build a compact kernel of commands that could easily be learned and mastered.

To develop this kernel, we studied over 40 algorithm animations built with the XTango system[Sta92]. The animations' topics included sorting, searching, graph, tree, string and graphics algorithms, as well as animations of matrix multiplication, fft, hashing, producer-consumer problems, etc. These animations were built by over 25 different people, so they were not biased to a particular person's design methodology.

The first step in this analysis was to determine which types of graphical objects are commonly used in algorithm animations, and also how the appearance of an object depends on the program it is representing. Although XTango supports a wide variety of different graphical objects, only lines (15 times), circles (11 times), rectangles (13 times) and text (17 times) commonly appeared. Other objects such as polygons or ellipses appeared only 0-2 times.

When one of these graphical objects is created in an algorithm animation, its appearance usually depends on the state of the underlying program and the values of variables within the program. For instance, the position, size (length, width, height, radius) and label (for text) all may depend upon program values.

For lines, we found that the position and size of the line are its two attributes that vary. Position was either predetermined (no program dependence), dependent upon the values of variables in the program, or relative to other existing animation objects. Line size was either predetermined or relative to other objects.

The rectangle attributes of position and size (width and/or height) varied across animations. Both position and size were either predetermined or program variable dependent. These specifications were the same for circles with the size attribute being the circle's radius.

Finally, text objects varied along position and text string attributes. Text position was commonly either predetermined or dependent upon the position of some other graphical object. (Text is often used to label other objects.) Text strings were either predetermined or dependent upon program variables.

In addition to individual graphical objects, many of the algorithm animations manipulated rows or columns of objects. These structures were commonly used to represent program arrays. In specifying a row of objects, designers would identify a bounding box inside of which the individual objects were placed. The number of objects was either predetermined or dependent upon the value of a variable. Often, one dimension of the objects, such as rectangles' heights for a row, varied according to the values of the variables in the array the row represented. Other attributes that varied were the structure's orientation (horizontal or vertical) and the spacing between individual objects.

Table 1 lists a summary of all these graphical objects along with their program dependent attributes.

XTango animations also include the capability to designate particular positions in the

Object	Attribute	Specification(s)
Line	Position	Predetermined
		Relative to another object
		Program variable dependent
	Size	Predetermined
		Relative to another object
Rectangle	Position	Predetermined
		Program variable dependent
	Width, height	Predetermined
		Program variable dependent
Circle	Position	Predetermined
		Program variable dependent
	Radius	Predetermined
		Program variable dependent
Text	Position	Predetermined
		Relative to another object
	String	Predetermined
		Program variable dependent
Object array	Position	Predetermined (bounding box)
	Number	Predetermined
	Size	Predetermined
		Program variable dependent

Table 1: Summary of graphical objects commonly used in XTango algorithm animations and how their attributes are specified. "Predetermined" means that the designer provided a value which remained constant.

display window. These positions often serve as the destination points of object movements. We found that this feature was very commonly used, so we included it in the constituent set of capabilities for the new system.

The second major step in identifying algorithm animation features was determining common actions or changes that objects underwent after they were created. In all the sample XTango animations examined, only five actions occurred more than a few times. The actions are

- Move an object to a particular position or to a position relative to another object.
- Change the color of an object.
- Change the fill style (outline or filled) of an object.
- Make an object flash
- Make two objects exchange positions (a special combination of movement actions).

After completing this survey, we organized these sets of graphical objects and common animation actions into a kernel of capabilities to be used as the basis for the graphical debugging system. Our intention was to allow designers to instantiate objects graphically and to select animation actions through a set of menu-based commands.

3 Interacting with Lens

In this section we describe how programmers interact with the Lens system. To begin a visual debugging session, a programmer issues the command, "lens foo", where *foo* is the name of an executable program. Lens immediately prompts the user with a list of source files to display. After the user chooses the initial file, Lens loads this source and awaits the entry of animation commands. (Other source files can be loaded via a menu command.) The entire Lens display appears as shown in Figure 2. The left area presents program source code, and the right area is a graphical editor for designing objects' appearances.

To specify how the program animation should look, a programmer chooses commands from the animation menu above the source code. It has seven options that correspond to the kernel of algorithm animation constituents found in our study of algorithm animations: 1) Create objects 2) Create location marker 3) Move 4) Fill 5) Color 6) Flash 7) Exchange.

When either of the first two commands is chosen, the programmer is prompted to enter the variable name of the object or location being depicted. This variable name is subsequently used to identify the object for the action commands. Lens then asks the programmer to use the graphical editor to design the object or location's appearance and/or position. This design is structured according to the object specifications that were discovered in the earlier study. For example, when a line is created, its position can be specified using the mouse, it can be specified relative to another object (by name or by picking a graphical object), or it can be specified to relate to the value of a program variable. All these choices are made via a dialog box entry or by graphical direct manipulation. Finally, Lens asks the

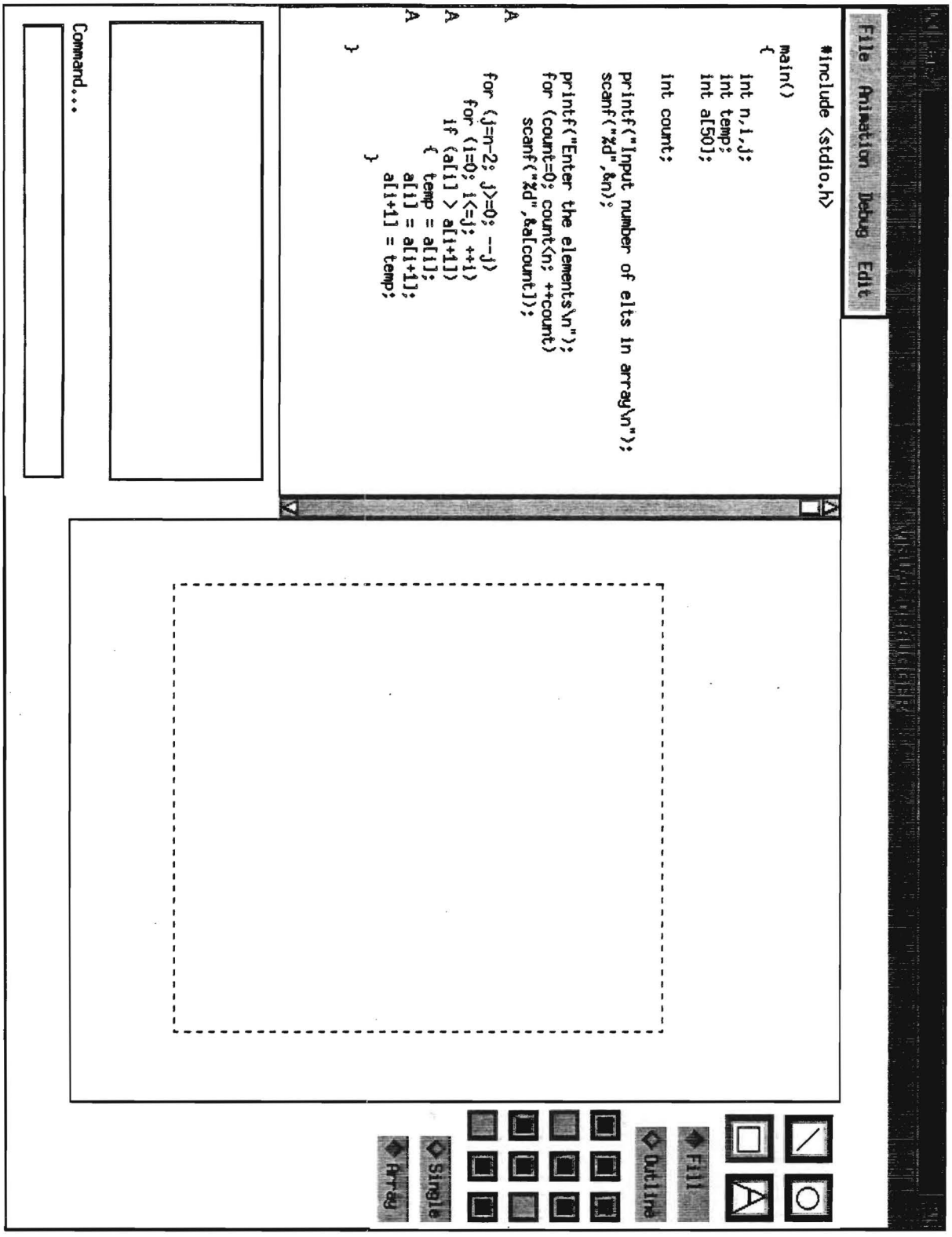

Figure 2: Lens display presented to a programmer. The left section shows source and the right section is the graphical editor.

user to click the mouse on the source code line at which object or location creation should occur. Lens indicates the presence of animation commands by showing an 'A' beside the code line.

When a programmer chooses one of the five action commands, Lens asks for the name of the variable, and consequently graphical object, to which the action should apply. The programmer simply types in a variable name such as `x` or `a[i]` and Lens notes the choice. Finally, the programmer must select the source code line on which to place the command. Multiple animation commands can be placed on single lines.

When the programmer wishes to execute the program and see its animation, s/he chooses the "Run" command from the Debug menu. Lens then pops up an animation window and displays the animation that corresponds to the programmer's design and this particular execution. Lens uses the routines from the XTango system to generate the animations it presents.

If the animation is not sufficient or not what the programmer wanted, s/he can go back, add or delete animation commands, and rerun the animation. Lens also supports saving animation commands between sessions, so that program debugging can be resumed at a later time using the same animation context.

Building a Sample Animation

Using Lens, it is straightforward to build program animations in minutes. Figure 2 shows the source and placement of three animation commands for building a bar-chart style bubblesort animation view commonly depicted in algorithm animation systems.

The first annotation is a *Create Object* command. The programmer created an appearance for the variable named `a`. He specified a rectangle array presentation, drew a bounding box for it, selected a horizontal orientation, and specified that the number of objects is dependent upon the value of the variable `n`. Finally, the programmer provided sample minimum and maximum values for the array elements. Lens requires this information to scale the heights of rectangles.

The second animation annotation corresponds to two animation commands, both of *Flash* type. The programmer specified that the objects corresponding to `a[i]` and `a[i+1]` be flashed to indicate a comparison in the program.

The final annotation corresponds to an *Exchange* command. The programmer specified that objects `a[i]` and `a[i+1]` be swapped. Lens will illustrate a smooth interchange motion for this command.

Figure 3 illustrates the view of this animation specification and a frame from the resulting animation. With a few more simple *Color* and *Fill* commands, the animation can be refined further.

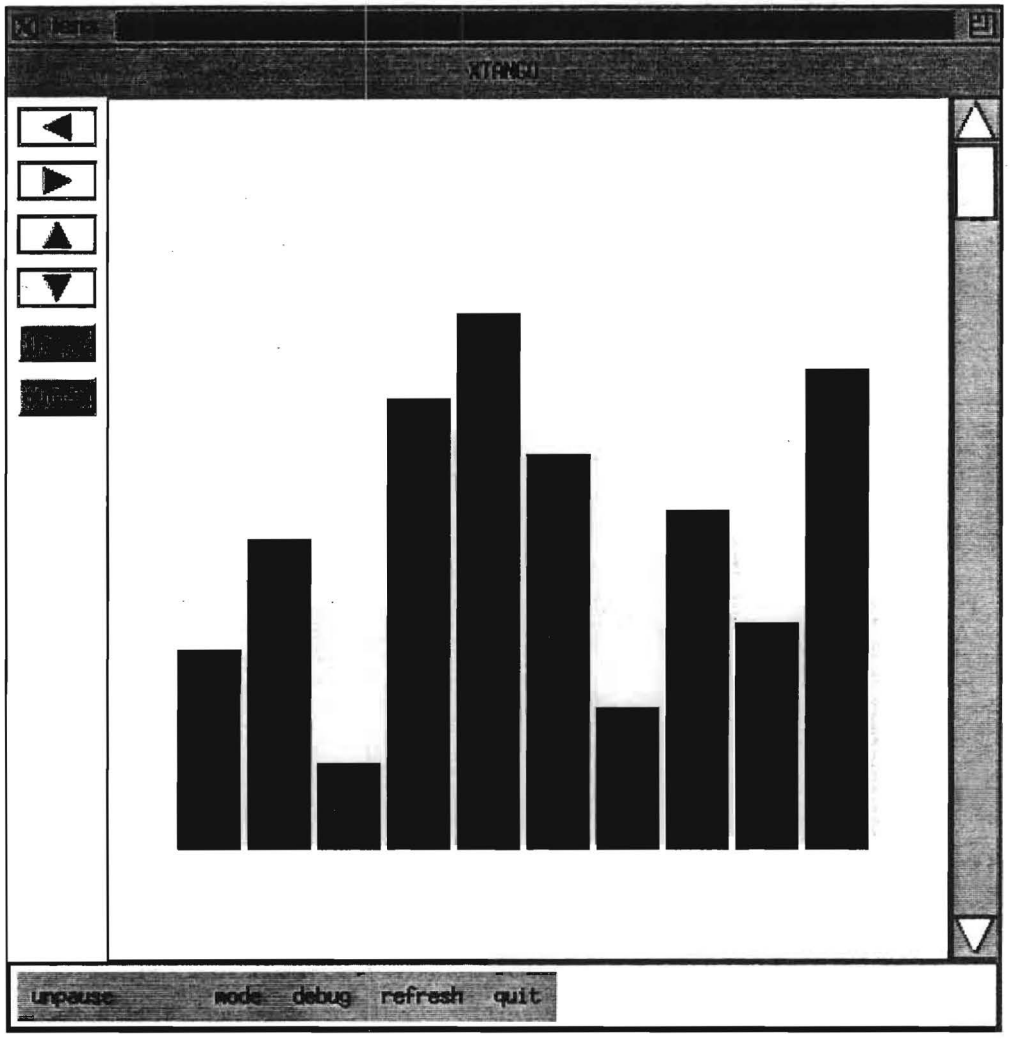

Figure 3: Frame from the bubblesort animation built using Lens and three animation commands. The buttons to the left provide panning and zooming. The scrollbar to the right controls the animation speed.

4 System Implementation

The creation of the Lens system presented a number of interesting implementation challenges. In this section we highlight a few of the most important ones.

4.1 Interaction with dbx

To acquire information about the program that is being run, Lens establishes a connection with the *dbx* debugger. The interface with dbx is similar to the approach used by the program *xdbx*. Lens communicates with dbx through a pseudo terminal that is a pair of master and slave devices: `/dev/pty??` and `/dev/tty??`. The pty is opened for both reading and writing. After a child process is created via `fork`, the child process closes the master side of pty, redirects stdin, stdout and stderr of dbx to pty, unbuffers output data from dbx, and `execs` dbx. The parent process closes the slave side of pty, sets the dbx file pointer to nonblocking mode, opens the file pointer to read/write access to dbx, sets line buffered mode, and then monitors the output from and passes input to dbx.

When the user commands Lens to create an animation action, the parent sends a *stop at* command to dbx. Later, when the program is executing and dbx stops at that line, the parent executes the animation action that was specified at the line. The parent also may acquire other information from dbx; for example, if the value of a variable is required, a *print* command is passed to dbx, and if the type of a variable is required, a *what is* command is passed to dbx. If dbx passes an output, the parent processes it and takes the appropriate action. For example, if the program that is being debugged has an error and dbx sends an error message, the parent will display the error message for the user and halt the execution of the program. If dbx sends an output which the parent does not recognize, the parent assumes that the output is from the program itself and outputs it for the user to see. Thus, the overall structure of the Lens system is shown in Figure 4.

4.2 Specifying the Animation Actions

In order to make the specification of animation actions as easy as possible for the programmer, Lens requires some subtle internal manipulations. For example, when a programmer types in the target for a command such as *Color*, s/he simply enters a text string such as `b1` at a dialog prompt. Lens resolves this entry into an internal database of objects and locations that already has been created. Lens also must alert the programmer to syntactic errors made at this level.

If an object's attribute is dependent on a program variable, the system asks the user to specify its maximum and minimum value in order for Lens to scale the object appropriately during the actual animation. Another point worth mentioning is that when the user chooses a variable, the system checks the type of that variable and rejects the animation action if the variable is not of the appropriate type. For example if a position is dependent on a variable, the variable must be an integer, double, short, long or float.

Lens also must be flexible in its interpretation of animation actions. If a programmer chooses a *Move* command, s/he can specify the name of another program entity which

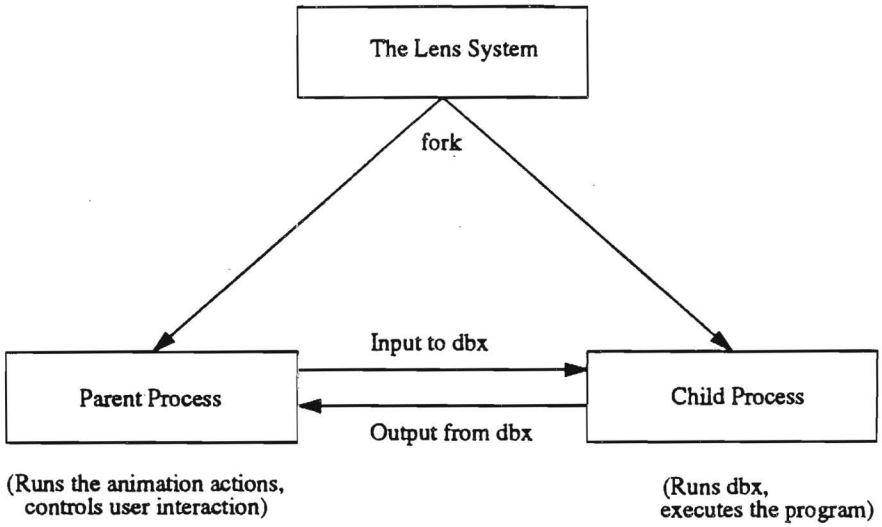

Figure 4: Lens system configuration illustrating how the various processes communicate with each other.

should be the destination of the movement action. This entity can be either a location or a graphical object. Lens must make this determination, and if it is an object, use the object's current position as the destination point.

The *Exchange* animation requires the user to specify the variables for the two objects (they cannot be locations) that are to be exchanged. The *Flash*, *Change Fill* and *Change Color* animations require the user to specify the variable for the image on which the action is to take place. The *Change Color* animation also requires the user to select a color from a palette of colors that is displayed. All these animation actions may be applied to an object or location that is an array element. For example, the user may specify that s/he wants to exchange `a[i]` and `a[i+1]`. To handle this situation, Lens must first check whether there is a '[' in the variable name. If so, it checks to see if the variable specified before the '[' refers to an array. If not, it will signify an error and reject that animation. Otherwise, during the actual animation execution, it will get the value of the array index (it may be a constant, variable or a simple expression) and use that index to acquire the appropriate graphical object or location.

4.3 Executing the Animations

After the programmer has built all the animations and wants to run the program, Lens must dispatch a *run* command to dbx. Before it does that, however, Lens goes through the list of animations (the animations are kept in a linked list) and sends a *stop at* command to dbx for each animation at the appropriate line. One unique feature of dbx is its assignment of line numbers to the logical file that are different from the actual line numbers of the source text file. This is due to blank lines and statements stretching over more than one line. Therefore, if one passes a *stop at n* command to dbx, the line number that dbx actually stops at may be different from *n*. Fortunately, dbx returns the line number where it will

stop given a particular *stop at* request. Lens uses this value and stores it for subsequent specifications.

When dbx stops at a particular break point and send a message to Lens, the system scans the list of animation commands to find out the command(s) which caused the break point. When it finds the command(s), it executes the action(s) specified there. For this, it may need to send other messages to dbx, for example, *print i* if the attribute of an object, such as its width, depends on the variable `i`.

It is possible that the program being debugged requires some input from the user via stdin. In Lens, dbx gets all its input from the parent process which was established. Hence, the parent must set its input to non-blocking mode and constantly poll the external input buffer. If there is ever any user input, Lens passes it on to dbx.

5 Future Plans

All of the capabilities described in this article have been implemented and are functional. Lens is currently running on Sun workstations under X11 and Motif. Our future plans involve continued development and refinement of Lens and its interface. In particular, we hope to

- Improve the somewhat "clunky" user interface for specifying object appearances and binding these representations to program entities. Currently, much of this interaction occurs through dialog boxes. We plan to utilize more of a direct manipulation[Shn83] approach. Two possibilities for improvement are choosing program variables by selecting them with the mouse from the program text, and specifying graphical "connection links" between graphical attributes (height, size, position) and program variables.
- Allow users to browse and examine animation commands that have been registered in the program source. Currently, we simply indicate the presence of animation commands by the 'A' annotation.
- Provide full dbx command capabilities to the user so that s/he can interact with the execution more.
- Add traditional data structure display capabilities to the system for further debugging support.
- Perform user testing to examine the usability of Lens' interface, and most importantly, to better understand how Lens can be utilized in program debugging.

Currently, our work with Lens is at the proof of concept phase. We wanted to address the challenge of building a program animation system that requires no coding nor any knowledge of a graphics paradigm, and that is integrated with a system debugger. We believe that Lens meets this challenge. Now, we must examine the system and understand its strengths and weaknesses as a debugging and tracing aid. This fall we plan to use Lens in algorithm design class to allow students to build their own program animations. We expect Lens to have utility as a learning aid in addition to program development. We also

expect to make the system available via anonymous ftp soon, and we hope that feedback from its use along with future empirical studies helps us to better understand the possible role of visualization and animation in program development and debugging.

Acknowledgments

This work supported in part by the National Science Foundation under contract CCR-9109399.

References

[Bae86] Ronald M. Baecker. An application overview of program visualization. *Computer Graphics: SIGGRAPH '86*, 20(4):325, July 1986.

[Bas85] David B. Baskerville. Graphic presentation of data structures in the DBX debugger. Technical Report UCB/CSD 86/260, University of California at Berkeley, Berkeley, CA, October 1985.

[BK91] Jon L. Bentley and Brian W. Kernighan. A system for algorithm animation. *Computing Systems*, 4(1), Winter 1991.

[Bro88] Marc H. Brown. Perspectives on algorithm animation. In *Proceedings of the ACM SIGCHI '88 Conference on Human Factors in Computing Systems*, pages 33–38, Washington D.C., May 1988.

[BS81] Ronald M. Baecker and David Sherman. Sorting Out Sorting. 16mm color sound film, 1981. Shown at SIGGRAPH '81, Dallas TX.

[BS85] Marc H. Brown and Robert Sedgewick. Techniques for algorithm animation. *IEEE Software*, 2(1):28–39, January 1985.

[Dui86] Robert A. Duisberg. Animated graphical interfaces using temporal constraints. In *Proceedings of the ACM SIGCHI '86 Conference on Human Factors in Computing Systems*, pages 131–136, Boston, MA, April 1986.

[Dui87] Robert A. Duisberg. Visual programming of program visualizations. A gestural interface for animating algorithms. In *IEEE Computer Society Workshop on Visual Languages*, pages 55–66, Linkoping, Sweden, August 1987.

[Gou75] J. D. Gould. Some psychological evidence on how people debug computer programs. *International Journal of Man-Machine Studies*, pages 151–182, 1975.

[HWF90] Robert R. Henry, Kenneth M. Whaley, and Bruce Forstall. The University of Washington illustrating compiler. *Sigplan Notices: SIGPLAN '90*, 25(6):223–233, June 1990.

[ISO87] Sadahiro Isoda, Takao Shimomura, and Yuji Ono. VIPS: A visual debugger. *IEEE Software*, 4(3):8–19, May 1987.

[KA87] Irving Katz and John Anderson. Debugging: An analysis of bug location strategies. *Human-Computer Interaction*, 3(4):351–399, 1987.

[Mye83] Brad A. Myers. A system for displaying data structures. *Computer Graphics: SIGGRAPH '83*, 17(3):115–125, July 1983.

[Mye90] Brad A. Myers. Taxonomies of visual programming and program visualization. *Journal of Visual Languages and Computing*, 1(1):97–123, March 1990.

[PSB92] Blaine A. Price, Ian S. Small, and Ronald M. Baecker. A taxonomy of software visualization. In *Proceedings of the 25th Hawaii International Conference on System Sciences*, volume II, pages 597–606, Kauai, HI, January 1992.

[Rei85] Steve P. Reiss. Pecan: Program development systems that support multiple views. *IEEE Transactions on Software Engineering*, SE-11(3):276–285, March 1985.

[Shn83] Ben Shneiderman. Direct manipulation: A step beyond programming languages. *Computer*, 16(8):57–69, 1983.

[SI91] Takao Shimomura and Sadahiro Isoda. Linked-list visualization for debugging. *IEEE Software*, 8(3):44–51, May 1991.

[SP92] John T. Stasko and Charles Patterson. Understanding and characterizing software visualization systems. In *Proceedings of the IEEE 1992 Workshop on Visual Languages*, Seattle, WA, September 1992.

[Sta90] John T. Stasko. TANGO: A framework and system for algorithm animation. *Computer*, 23(9):27–39, September 1990.

[Sta91] John T. Stasko. Using direct manipulation to build algorithm animations by demonstration. In *Proceedings of the ACM SIGCHI '91 Conference on Human Factors in Computing Systems*, pages 307–314, New Orleans, LA, May 1991.

[Sta92] John T. Stasko. Animating algorithms with XTANGO. *SIGACT News*, 23(2):67–71, Spring 1992.

C-50-661
2

NSF Grant Conditions (Article 17, GC-1, and Article 9, FDP-11) require submission of a Final Project Report (NSF Form 98A) to the NSF program officer no later than 90 days after the expiration of the award. Final Project Reports for expired awards must be received before new awards can be made (NSF Grants Policy Manual Section 677).

Below, or on a separate page attached to this form, provide a summary of the completed projects and technical information. Be sure to include your name and award number on each separate page. See below for more instructions.

PART II - SUMMARY OF COMPLETED PROJECT (for public use)

The summary (about 200 words) must be self-contained and intelligible to a scientifically literate reader. Without restating the project title, it should begin with a topic sentence stating the project's major thesis. The summary should include, if pertinent to the project being described, the following items:

The primary objectives and scope of the project
The techniques or approaches used only to the degree necessary for comprehension
The findings and implications stated as concisely and informatively as possible

See attached

PART III - TECHNICAL INFORMATION (for program management use)

List references to publications resulting from this award and briefly describe primary data, samples, physical collections, inventions, software, etc. created or gathered in the course of the research and, if appropriate, how they are being made available to the research community. Provide the NSF Invention Disclosure number for any invention.

See attached

Certify to the best of my knowledge (1) the statements herein (excluding scientific hypotheses and scientific opinion) are true and complete, and the text and graphics in this report as well as any accompanying publications or other documents, unless otherwise indicated, are the original work of the signatories or of individuals working under their supervision. I understand that willfully making a false statement or concealing a material fact in this report or any other communication submitted to NSF is a criminal offense (U.S. Code, Title 18, Section 1001).

[illegible]	6/6/94
Principal Investigator/Project Director Signature	Date

IMPORTANT
MAILING INSTRUCTIONS
Return the *entire* packet plus all attachments in the envelope attached to the back of this form. Please copy information from Part I - Block 1 to the *Attention* block on the envelope.

NSF Form 98A (Rev. 6/92)

PART II – SUMMARY OF COMPLETED PROJECT

This project examined how visualization and animation techniques could be incorporated into a source code debugging tool to foster visual debugging. Our research sought methods to allow programmers to build algorithm animation style program views of their own programs without resorting to textual graphics coding. To begin the work, we identified a kernel of graphical objects (line, circle, rectangle, text) and animation actions (move, color, flash, fill, exchange, delete) that were commonly used in algorithm animations and that would be used as the basis for the graphical presentations. Next, we developed methodologies to provide these capabilities to programmers without forcing the programmers to do textual graphics coding. Finally, we implemented a prototype system called Lens that is the manifestation of the concepts we developed. Lens allows programmers to easily define mappings from their programs into a visualization. It is integrated with the dbx debugger, and it provides XTango algorithm animation views.

PART III – TECHNICAL INFORMATION

References:

John T. Stasko and Charles Patterson. "Understanding and characterizing software visualization systems" In *Proceedings of the 1992 IEEE Workshop on Visual Languages*, pages 3-10, Seattle, WA, September 1992.

Sougata Mukherjea and John T. Stasko. "Applying algorithm animation techniques for program tracing, debugging, and understanding" In *Proceedings of the 15th International Conference on Software Engineering*, pages 456-465, Baltimore, MD, May 1993.

Sougata Mukherjea and John T. Stasko. "Toward Visual Debugging: Integrating Algorithm Animation Capabilities within a Source Level Debugger" submitted to *ACM Transactions on Computer-Human Interaction.*

The software system Lens was created as a result of this project. Lens is being made available to the community via anonymous ftp. It is available as the file `pub/lens.tar.Z` on the machine `par.cc.gatech.edu`.

Further information about this project and the Lens system can be found in the accompnaying Technical Report.

Final Project Report
Accompanying Technical Report
June 6, 1994
NSF RIA Award CCR-9109399

Adapting Algorithm Animation Techniques for Program Debugging and Testing

John T. Stasko

College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280

In my dissertation work, I developed a system for building algorithm animations, which are dynamic graphical depictions of computer algorithms that are used to help teach students how the algorithms work. After completing my dissertation, I speculated that these kinds of algorithm animation techniques might be useful for program testing and debugging as well.

This Research Initiation Award helped me study how algorithm animation techniques could be integrated with a system debugger to give programmers graphical depictions of how their programs are working. The research began by evaluating existing algorithm animations to find a kernel of important graphical presentations and operations. From there, we integrated these operations within a system debugger (dbx) and we provided a convenient graphical user interface for access to these operations. That allowed programmers to build animations of their programs without doing any extra textual coding.

Our work resulted in the implementation of the Lens system. Lens has been made available to other researchers and developers via anonymous ftp as the file `pub/lens.tar.Z` on the machine `par.cc.gatech.edu`.

Some of our thoughts on algorithm animation and software visualization that resulted from early work on this project are described in [2]. Early results of our research on the Lens system are described in the publication [1]. We also currently have a journal paper in submission to *ACM Transactions on Computer-Human Interaction* about the project. Below is the abstract from this article:

Much of the recent research in software visualization has been polarized toward two opposite domains. In one domain that we call *data structure* and *program visualization*, low-level canonical views of program structures are generated automatically. These types of views, which do not require programmer input or intervention, can be useful for testing and debugging software. Often, however, their generic, low-level views are not expressive enough to adequately convey how a program functions. In the second domain called *algorithm animation*, designers hand-craft abstract, application-specific views that are useful for program understanding and teaching. Unfortunately, since algorithm animation development typically requires time-consuming design with a graphics package, it will not be used for debugging, where timeliness is a necessity. However, we speculate that the application-specific nature of algorithm animation views could be a valuable debugging aid for software developers as well, if only the views could be easy and rapid to create. We have developed a system called Lens that occupies a unique niche between the two domains discussed above and explores the capabilities that such a system may offer. Lens allows programmers to rapidly (in minutes) build algorithm animation-style program views without requiring any sophisticated graphics knowledge and without using textual coding. Lens also is integrated with a system debugger to promote iterative design and exploration.

References

[1] Sougata Mukherjea and John T. Stasko. Applying algorithm animation techniques for program tracing, debugging, and understanding. In *Proceedings of the 15th International Conference on Software Engineering*, pages 456–465, Baltimore, MD, May 1993.

[2] John T. Stasko and Charles Patterson. Understanding and characterizing software visualization systems. In *Proceedings of the 1992 IEEE Workshop on Visual Languages*, pages 3–10, Seattle, WA, September 1992.